MS.NETGrid OGSI User Manual

Project Title: MS.NETGrid

Document Title: MS.NETGrid OGSI User Manual

Document Identifier: MS.NETGrid-OGSIUserManual-v2.0

Authorship: Ally Hume, Daragh Byrne

Document History:

Personnel	Date	Summary	Version
EPCC	17th March 2004	EPCC Approved	2.0
EPCC	30th January 2004	EPCC Approved	1.2
EPCC	16th June 2003	EPCC Approved	1.0
MJJ	13th June 2003	Second Draft – Changes and Comments	0.2
ACH	10th June 2003	First Draft	0.1

Approval List: *EPCC: Project Leader, Technical Staff, Technical Reviewer, Coach*

Mike Jackson, Daragh Byrne, Ally Hume, Ali Anjomshoaa, Neil Chue Hong

Approval List: *Microsoft Research Limited: Managing Director, University Relations x 3*

Andrew Herbert, Fabien Petitcolas, Van Eden, Dan Fay

Copyright © 2004 The University of Edinburgh. All rights reserved.

Contents

Contents 2

1 Introduction 4

1.1 Product Licence 4

1.2 Project WWW Site 4

1.3 Support and Queries 4

2 Installation 5

2.1 Configuration 5

3 Running a Demonstration Client 6

4 Class documentation 8

5 Writing a simple service 9

5.1 Service Lifetime 9

5.2 Developing a service 9

5.2.1 Write the service class 9

5.2.2 Write the proxy class 10

5.2.3 Compile the service class and proxy class into an assembly 11

5.2.4 Write a `.asmx` file for the service 11

5.2.5 Configure the OGSI container 11

5.2.6 Installing the service 12

6 Writing a client of a Grid Service 13

7 Writing a client managed service and a Factory service 14

7.1 Write the client managed service class 14

7.2 Write the factory service class 15

7.3 Write factory creator class 16

7.4 Write proxy classes 17

7.5 Compile classes into an assembly 18

7.6 Write `.asmx` files for the transient service and factory service 18

7.7 Configure the OGSI container 18

7.8 Install the services 19

8 Writing a client of a container managed service 20

9 Service data 22

9.1 Default service data elements 22

9.2 Service data set 22

9.3 The `ServiceData` class 23

9.3.1 Service data element properties 23

9.3.2 Service data values 23

9.3.3 Using a callback to determine service data values dynamically 24

9.3.4 Lifetime attributes 24

9.3.5 Service data evaluators 25

10 PortTypes 26

10.1 OgsiPortTypeAttribute 26

11 Notification 28

11.1 Making a service data element notifiable 28

11.2 Creating a Source Service 28

11.3 Creating a sink service 29

11.3.1 The debug notification service 30

11.4 Writing clients of notification sources 30

12 Persisting services 32

12.1 Configuring the container 32

13 Example services 33

14 Glossary 34

References 36

1 Introduction

MS.NETGrid is an ASP.NET-based implementation of the Open Grid Services Infrastructure. This user manual is aimed at the Grid Service developers and describes how to use the MS.NETGrid implementation to create Grid Services. Details of how to write clients that use these services – in the form of running examples – are also included. This document complements [*MS.NETGridOgsiDesignOverview*] that is included with this distribution, available from http://www.epcc.ed.ac.uk/~ogsanet/software.html. Further information about the use of this software is available via http://www.epcc.ed.ac.uk/~ogsanet/course4.

1.1 Product Licence

The MS.NETGrid software is released under the licence described in LICENSE.TXT under the root directory of the distribution.

1.2 Project WWW Site

The MS.NETGrid Project WWW site is at:

http://www.epcc.ed.ac.uk/~ogsanet

1.3 Support and Queries

Please note that our software is intended as a proof-of-concept to demonstrate the applicability of Microsoft .NET in general and ASP.NET in particular to the development of Grid Services rather than as a fully-functioning product. This software is supported indirectly by the UK e-Science Grid Support Centre and queries should be directed to:

ogsanet-queries@epcc.ed.ac.uk

2 Installation

The download consists of a zip file called `MS.NETGridOGSI-2.0.zip`. Extract this to a convenient directory of your choice. We strongly recommend that you extract to the root directory of your hard drive, e.g. `C:\`. At this location, there will be a directory called `MS.NetGrid`. This directory will have a number of subdirectories one of which is called `Ogsi.Container`. This `Ogsi.Container` directory contains the ASP.NET-based OGSI container, as well as all source code for the container.

The `Ogsi.Container` directory must be set up as a virtual directory under Internet Information Services (IIS). This can be done using Microsoft's Internet Services Manager; see the user documentation for this product for details of creating a virtual directory. Call the virtual directory `ogsa`. A tutorial for setting up virtual directories is available via http://www.epcc.ed.ac.uk/~ogsanet/course4.

Throughout the rest of this document the notation *<OGSIContainer>* refers to the full path name of the `Ogsi.Container` directory, e.g. `C:\MS.NETGrid\Ogsi.Container`. *<install_dir>* refers to the directory into which the `MS.NETGrid` directory was unzipped, e.g. `C:\`.

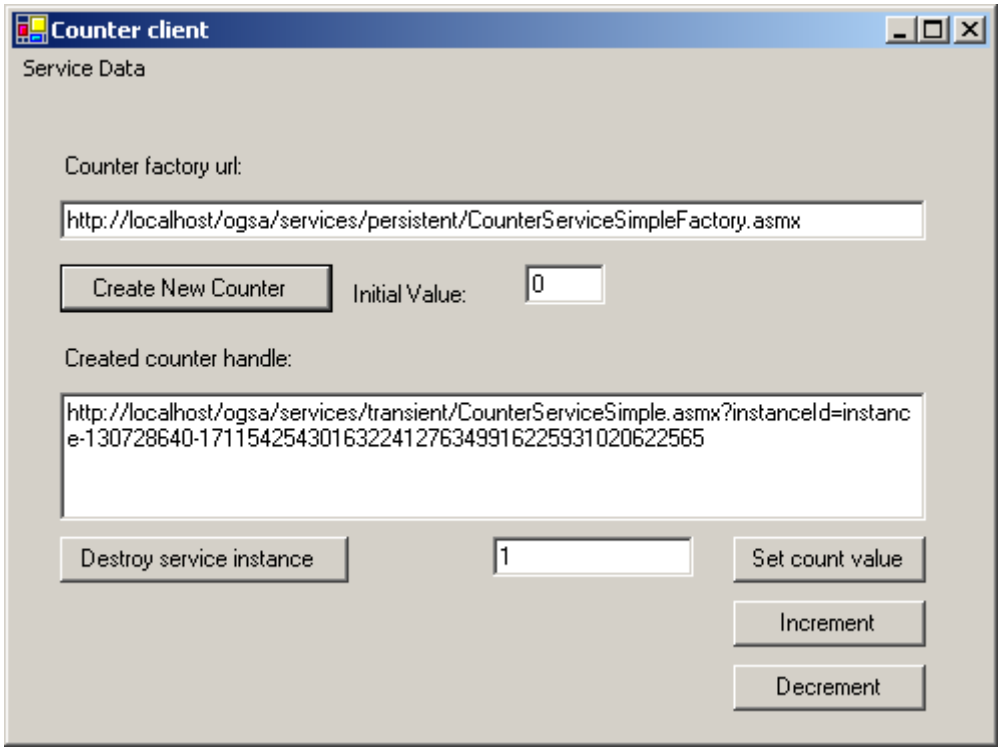
2.1 Configuration

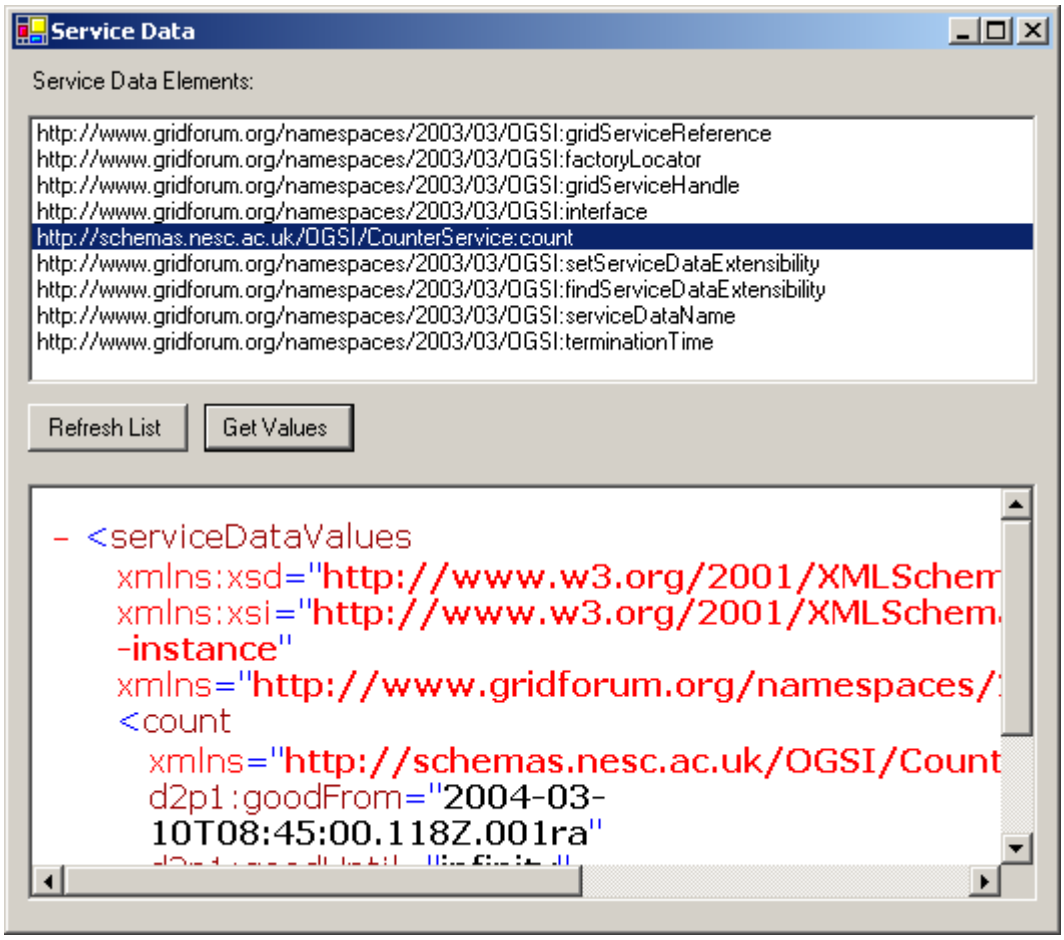
In the *<OGSIContainer>* directory is a file called `Web.config`. This file contains information used to configure the MS.NETGrid software. Open the file with a text editor. There are several places in this file where the path `D:\MS.NETGrid\…` can be seen. Please replace the string `D:\MS.NETGrid\` with the string *<install_dir>*`\MS.NETGrid`, depending on where you installed the software. The software is unlikely to work correctly if you neglect to carry out this step.

3 Running a Demonstration Client

The installation is configured to initialise a counter service factory that can be used in conjunction with a supplied client to test your installation. The client is located in `<install_dir>/MS.NETGrid/Ogsi.Container/demoservices/CounterService/CounterClient`. Go to the `bin` subdirectory and run the `CounterClient` executable. Click on the "Create New Counter" button. This creates a new instance of a transient counter service. The Grid Service Handle for this service instance should appear in the "Created counter handle" box. If it does not, please make sure that IIS is running and the `Ogsi.Container` directory is visible from Internet Services Manager.

Several operations may be performed on this service instance. The count value may be set using the "Set count value" button. This sets the current value of the counter to be that of the box to the left of this button. The "Increment" and "Decrement" buttons may be used to adjust the value of the counter by the amount displayed in the box between the buttons. The "Service Data" menu can be used to obtain service data by name. The ServiceData values are displayed as pure XML. If this client operates correctly, the installation has been successful. The correctly operating client looks like the following screenshot.




Service Data
Service Data Elements:
http://www.gridforum.org/namespaces/2003/03/OGSI:gridServiceReference
http://www.gridforum.org/namespaces/2003/03/OGSI:factoryLocator
http://www.gridforum.org/namespaces/2003/03/OGSI:gridServiceHandle
http://www.gridforum.org/namespaces/2003/03/OGSI:interface
http://schemas.nesc.ac.uk/OGSI/CounterService:count
http://www.gridforum.org/namespaces/2003/03/OGSI:setServiceDataExtensibility
http://www.gridforum.org/namespaces/2003/03/OGSI:findServiceDataExtensibility
http://www.gridforum.org/namespaces/2003/03/OGSI:serviceDataName
http://www.gridforum.org/namespaces/2003/03/OGSI:terminationTime
Refresh List
Get Values
- <serviceDataValues
xmlns:xsd="http://www.w3.org/2001/XMLSchem
xmlns:xsi="http://www.w3.org/2001/XMLSchem
-instance"
xmlns="http://www.gridforum.org/namespaces/
<count
xmlns="http://schemas.nesc.ac.uk/OGSI/Count
d2p1:goodFrom="2004-03-
10T08:45:00.118Z.001ra"

4 Class documentation

Source code documentation is available in Microsoft Help format at:

`<install_dir>/MS.NETGrid/Doc/API/MS.NETGridHelp.chm`

Simply double click on this file to start help.

5 Writing a simple service

In this section we go through the stages required to write and deploy a simple Grid Service.

5.1 Service Lifetime

In the MS.NETGrid software, in common with other OGSI implementations, there are two types of service lifetime. A **container managed** Grid Service is one whose lifetime is managed by the OGSI container. These services are created when the OGSI container starts up. **Client managed** Grid Services are services that are created by OGSI factory services and whose lifetime is soft-state managed by the client. In version 2.0 of MS.NETGrid-OGSI, unlike previous versions, it is possible to write a service once and use it in either a container managed OR a client managed manner.

5.2 Developing a service

The development stages for MS.NETGrid Grid Services are:

1. Write the service classes that implement the functionality of the service.
2. Write the proxy class that exposes the service's public methods via ASP.NET.
3. Compile the service class and proxy class into an assembly.
4. Write a `.asmx` file for the service.
5. Configure the OGSI container to be aware of your new service.
6. Install the service.

The Grid Service we write here will be a container managed Grid Service. We will implement a single method called `hello()`. This method takes one argument called `name` and will return the string "`hello` *`name`* *`n`*" where *`name`* is the value of the `name` argument and *`n`* is an integer that starts at 1 and increments every time the method is called.

The Grid Service will inherit additional methods (e.g. `findServiceData()` and `destroy()`) that correspond to the GridService portType operations that all Grid Services must provide.

5.2.1 Write the service class

The service class is the class that implements the specific functionality of the Grid Service. The code for this class is as follows and should be saved in a file called `MyServiceImpl.cs`:

```
using Ogsi.Core;
namespace MyGridServices
{
  public class MyServiceImpl : GridServiceSkeleton
  {
    private int count_ = 1;

    public string hello( string name )
```

```
    {
      return "hello " + name + " " + (count_++).ToString();
    }
  }
}
```

This class inherits from the `Ogsi.Core.GridServiceSkeleton` class. All Grid Services inherit from this class, regardless of whether they are container managed or client managed services.

As can be seen from the code, implementing the service class for a Grid Service is very similar to writing any other class. The `Ogsi.Core.GridServiceSkeleton` class gives access to other Grid Service infrastructure such as service data that the service may wish to use. This simple example does not use service data.

5.2.2 Write the proxy class

The proxy class is used to expose the service's public methods, or, in other words, the operations of the Grid Service, to clients, using standard Web service protocols.

The code for the proxy class – which should be saved in a file called `MyService.cs` – is as follows:

```
using System.Web.Services;
using Ogsi.Container.Instances.Proxy;

namespace MyGridServices
{
  public class MyService : GridServiceInstanceAspProxy
  {
    [WebMethod]
    public string hello( string name )
    {
      object [] args = { name };
      return (string) CallMethodOnPortType(
         ContainerConstants.GRID_SERVICE_PORT_TYPE_PROVIDER,
         "hello",
         args
      );
    }
  }
}
```

This class inherits from the `GridServiceInstanceAspProxy` class (in the `Ogsi.Container.Instances.Proxy` namespace). The `System.Web.Services.WebMethodAttribute` is used to tell ASP.NET to expose this method as a public Web method of the service. Other ASP.NET method attributes may be used to provide further customisation, for example security. See the ASP.NET user documentation for details.

Every method that is to be exposed as part of the Grid Service must have a corresponding method defined for it in the proxy class. The methods in the proxy class are all very similar. They simply put the arguments into an array and then pass these arguments and the method name to the `CallMethod()` method that is a public method of the base class.

5.2.3 Compile the service class and proxy class into an assembly

To compile the service class and proxy class into an assembly use the following command line instruction (which should be typed on a single line):

```
csc /t:library /out:MyServices.dll MyServiceImpl.cs MyService.cs
    /r:<OGSIContainer>\bin\Ogsi.Container.dll
```

<OGSIContainer> is the path to the OGSI container installation.

If you are using the Microsoft Visual Studio development environment configure your project to output an assembly called `MyServices.dll`. You will have to add the following references to the project before building:

- *<OGSIContainer>*`\bin\Ogsi.Container.dll`
- `System.Web.Services.dll`

Regardless of whether you have used Visual Studio or the command line you should now have created an assembly called `MyServices.dll`, located in the current directory. You need to copy this `dll` to `<OGSIContainer>\bin`.

5.2.4 Write a `.asmx` file for the service

To utilise ASP.NET to deploy our Grid Service we must create a `.asmx` file for the service. This simply maps the service name to the proxy class.

The `.asmx` file is (save this in a file called `MyService.asmx`):

```
<%@ WebService Class="MyGridServices.MyService"%>
```

The name of the `.asmx` file determines the Grid Service Handle (GSH) for the service.

Copy the `.asmx` file you have just written to the `<install_dir>/Ogsi.Container/services` directory.

5.2.5 Configure the OGSI container

For each type of Grid Service hosted by the OGSI container an entry must be added to the container's `Web.config` file (this file is located at *<OGSIContainer>*`\Web.config`).

Inside the `<gridContainer.config/serviceTypeDescriptors>` element of the file add the following entry:

```
<serviceType
      typeId=”MyService”
      skeletonClass=”MyGridServices.MyServiceImpl, MyServices”
     proxyPath=”ogsa/services/MyService.asmx
/>
```

This entry defines the `HelloService` service type within the container. It also specifies the class that forms the implementation of the service.

Having defined the `HelloService` serviceType within the container, we now wish to deploy a container managed instance of this service within the container. Add the following entry to the `gridContainer.config/containerManagedServices` element of

Web.config.

```
<service type="MyService" serviceName="MyFirstService" />
```

Note that the value in the `type` attribute of this element corresponds to the `typeId` value for the service type descriptor.

If in doubt about any of these steps follow the example services that are already deployed in `Web.config`.

5.2.6 Installing the service

To install the service:

- Copy the `MyServices.dll` assembly to *<OGSIContainer>*`\bin`.
- Copy the `MyService.asmx` file to *<OGSIContainer>*`\services\persistent`.
- Restart IIS.
    - To restart IIS obtain the control panel, select Administrative Tools and then select Internet Services Manager. Select your server in the tree view on the left hand side, right click and select `All Tasks -> Restart IIS`.

6 Writing a client of a Grid Service

Writing a client to talk to a Grid Service is identical to writing a client to talk to a Web Service. Firstly we must create a client side proxy class for the service using the appropriate Microsoft tools, and then use the generated class in our own application code.

To create a proxy class create a new directory and within that directory use the following command line instruction:

```
WSDL
http://localhost/Ogsi.Container/services/persistent/MyService.asmx?instanceId=MyFirstService
```

If you are accessing our container from a remote host then you can replace localhost with the name of the host upon which our container is running.

If the WSDL command does not work then you have not installed the full Microsoft .NET Software Development Kit. This software is available via http:// msdn.microsoft.com and should be installed before proceeding. If you have installed the software and still WSDL does not work, you need to set your PATH environment variable to point to the directory in which the file WSDL.exe resides.

Once successful, running WSDL in the above manner will create a proxy class called `MyService.cs`. We can now write a client that uses the service. Code for an example client of our service is:

```
using System;

class MyClient
{
   static void Main()
   {
      MyService GridService = new MyService();
      Console.WriteLine( GridService.hello( "Horace" ) );
   }
}
```

Compile this file along with the `MyService.cs` file as shown in the following command line instruction:

```
csc /out:MyClient.exe MyClient.cs MyService.cs
```

This will produce an executable called `MyClient.exe` that will call the Grid Service's `hello()` method when it is run producing the output shown below:

```
C:\Projects\ogsanet\client>MyClient
hello Horace 1

C:\Projects\ogsanet\client>MyClient
hello Horace 2

C:\Projects\ogsanet\client>MyClient
hello Horace 3
```

7 Writing a client managed service and a Factory service

In this section we go through the stages required to write and deploy a simple transient Grid Service with a corresponding persistent factory service. The factory service can be used to create instances of the transient Grid Service. The stages are:

1. Write the service class that implements the functionality of the client managed service.
2. Write the factory service class.
3. Write the factory initializer class that is used by the factory to correctly initialize instances of the transient service.
4. Write proxy classes for the service class and factory class.
5. Compile classes into an assembly.
6. Write `.asmx` files for the transient service and factory service.
7. Configure the factory and service classes.
8. Install the services.

The Grid Service we write here is very simple. It is initialised with a single string argument called `greeting` provided to the factory service – via the Factory portType `CreateService` operation – that will create our client managed Grid Service and we will implement a single method called `hello()`. The method takes one argument called `name` and will return the string "*`greeting name n`*" where *`greeting`* is the value of the `greeting` argument the service was initialised with, *`name`* is the value of the name argument of the `hello()` method and *`n`* is an integer that starts at 1 and increments every time the method is called. The Grid Service will inherit additional methods (e.g. `findServiceData()` and `destroy()`) that correspond to the GridService portType operations that all Grid Services must provide.

7.1 Write the client managed service class

The service class is the class that implements the specific functionality of the Grid Service. The code for this class – which should be saved in a file called `MyClientManagedServiceImpl.cs`) – is as follows:

```
using Ogsi.Core;

namespace MyGridServices
{
  public class MyClientManagedServiceImpl :
GridServiceSkeleton
  {
    private int    count_     = 1;
    private string greeting_;

    public string Greeting()
    {
```

```
      set
      {
        greeting_ = value;
      };
    }

    public string hello( string name )
    {
      return greeting_ + " " + name + " " +
            (count_++).ToString();
    }
  }
}
```

Note that, as before, this class inherits from the `Ogsi.Core.GridServiceSkeleton` class.

The most interesting feature of this class is that it has a property called `Greeting` that sets the string used as a greeting. We will see how to set this property in section 7.3.

7.2 Write the factory service class

The factory service class is the class that implements the functionality of the (container managed) factory service. The code – which should be saved in a file called `MyClientManagedServiceFactoryImpl.cs` – is as follows:

```
using System;
using Ogsi.Core;

namespace MyGridServices
{
  [OgsiPortType(
      typeof(FactoryPortType),
      "http://localhost/MyClientManagedServiceFactoryNS",
      "MyClientManagedServiceFactoryPortType") ]
  public class MyClientManagedServiceFactoryImpl :
                                GridServiceSkeleton
  {
  }
}
```

As can be seen this is a very simple class. The class is for a persistent service so it inherits from `Ogsi.Core.GridServiceSkeleton`. The service is to be a factory so we have used the `OgsiPortType` attribute to specify that the service supports the Factory portType. We pass this attribute the type of the MS.NETGrid supplied `FactoryPortType` class as a parameter. When we use the `OgsiPortType` attribute to specify that a service supports the Factory portType the majority of the implementation is done for us – all we have to do is write an initializer class to initialize instances of the service the factory creates and then specify that the initializer class is associated with the factory. These tasks are covered in sections 7.3 and 7.7 respectively.

The second and third arguments to the `OgsiPortType` attribute specify the namespace and name of the portType. These values are not used in the current release but are intended to provide a reference to the portType for use in future releases.

7.3 Write factory creator class

The factory initializer class is used to correctly initialize instances of services. This mechanism is used as the service author does not have any direct control over the creation of the service instances – this is handled by the factory which uses a default constructor. The code – which should be saved in a file called `MyServiceInitializer.cs` – is:

```
using System;
using System.Xml;
using Ogsi.Core;

namespace MyGridServices
{
  public class MyServiceInitializer : IServiceInitialiser
  {
    public void InitialiseServiceFromParameters(
           GridServiceSkeleton instance,
           XmlElement creationParams
           )
    {
      string greeting = "hello";

      // creationParams is an XmlElement containing
      // parameters to be used in the creation of the new
      // service instance that may be passed by the client.
      if (creationParams  != null)
      {
        if (creationParams.Name != "initialValue")
        {
          throw new ArgumentException(
            "creationParams should be of type initialValue");
        }

        greeting = creationParams.ChildNodes[0].Value;
      }

      // Initialize the service object correctly. This can
       // be as complicated as you like.
       MyClientManagedServiceImpl service =
          (MyClientManagedServiceImpl) instance;
       service.Greeting = greeting;
    }
  }
}
```

The factory initializer class must implement the `IFactoryInitializer` interface. This interface has only one method called `InitializeServiceFromParameters()`.

The `creationParams` parameter of the `CreateServiceObject()` method can be used to pass an arbitrary XML element from the client to the factory. This allows you control over how your services' clients initialize the services. This is the case in the code above – the majority of the code is responsible for extracting the value for the service's `Greeting` property from an XML element called "`initialValue`".

7.4 Write proxy classes

Proxy classes are required to expose the services' public methods. We have to write proxy classes for both the greeting service and the factory service.

The code for the greeting service class's proxy class – which should be saved in a file called `MyClientManagedService.cs` – is:

```
using System.Web.Services;
using Ogsi.Container.Instances.Proxy;

namespace MyGridServices
{
  public class MyClientManagedService :
                        GridServiceInstanceAspProxy
  {
     [WebMethod]
     public string hello( string name )
     {
        object [] args = { name };
        return (string) CallMethodOnPortType(
            ContainerConstants.GRID_SERVICE_PORT_TYPE_PROVIDER,
            "hello",
            args);
     }
  }
}
```

This class is quite similar to the proxy class for the service discussed in section 5.2.2.

The code for the factory class's proxy class – which should be saved in a file called `MyClientManagedServiceFactory.cs` – is:

```
using System.Web.Services;
using Ogsi.Container.Instances.Proxy;
using Ogsi.Core.Types;

namespace MyGridServices
{
  public class MyClientManagedServiceFactory :
                      GridServiceInstanceAspProxy
  {

    [WebMethod]
    public LocatorType createService(CreationType creation)
    {
      object [] arguments = { creation };
      return (LocatorType) CallMethodOnPortType(
        "Ogsi.Core.FactoryPortType",
        "CreateService",
        arguments );
    }
  }
}
```

You will note again the use of fairly boilerplate code within the body of the proxy method.

Note that we pass the full name of the class implementing the `createService` method (i.e. `Ogsi.Core.FactoryPortType`) to `CallMethodOnPortType()`.

7.5 Compile classes into an assembly

To compile all the classes into an assembly use the following command line (this should be typed on a single line):

```
csc /t:library /out:MyClientManagedServices.dll
 MyClientManagedService.cs MyClientManagedServiceFactory.cs
 MyServiceInitializer.cs MyClientManagedServiceFactoryImpl.cs
 MyClientManagedServiceImpl.cs
 /r:<OgsiContainer>\bin\Ogsi.Container.dll
```

Where *<OGSIContainer>* is the path to the OGSI container installation.

7.6 Write `.asmx` files for the transient service and factory service

To utilise ASP.NET to deploy our Grid Service we must create a `.asmx` file for each of the client managed service and the factory service. As in the case of the persistent service this simply maps the service name to the proxy class.

The `.asmx` file for the transient service is (save in file `MyClientManagedService.asmx`):

```
<%@ WebService Class="MyGridServices.MyClientManagedService"%>
```

The `.asmx` file for the factory service is (save in file `MyClientManagedServiceFactory.asmx`):

```
<%@ WebService
Class="MyGridServices.MyClientManagedServiceFactory"%>
```

7.7 Configure the OGSI container

Service type descriptors must be added for each of the new service types you have created, as before. In the Web.config file (see section 5 for a reminder), place the following entries:

```
<serviceTypeDescriptors>
  … … <!--other service type descriptors -->
  <serviceType
    typeId="MyClientManagedService"
    skeletonClass="MyGridServices.MyClientManagedServiceImpl,
      MyClientManagedServices"
    proxyPath="ogsa/services/MyClientManagedService.asmx">
  </serviceType>
  <serviceType
    typeId=" MyClientManagedServiceFactory"

skeletonClass="MyGridServices.MyClientManagedServiceFactoryImp
l,
       MyGridServices"

proxyPath="ogsa/services/MyClientManagedServiceFactory.asmx">
  </serviceType>
```

This contents of this element are similar to that shown in section 5.2.5 for the simple

persistent service. Notice the value of the "persistence" attribute is now "transient".

In order to deploy and use an instance of the factory service, so that it can be used to create instances of the client managed service, we need to add the following to the `gridContainer.confg/containerManagedServices` section of `Web.config`:

```
<containerManagedServices>
  <service type="MyClientManagedServiceFactory"
           name="MyFirstFactory"></service>
.. .. ..
<containerManagedServices>
```

7.8 Install the services

To install the service:

- Copy the `MyClientManagedServices.dll` assembly to *<OGSIContainer>*`\bin.`
- Copy the `MyClientManagedService.asmx` file to *<OGSIContainer>*`\services`
- Copy the `MyClientManagedServiceFactory.asmx` file to *<OGSIContainer>*`\services`.
- Restart IIS as above.

8 Writing a client of a factory service

Writing a client to talk to a factory Grid Service is virtually identical to writing a client to talk to a Web Service. Firstly we must create a proxy classes for the services and then use them in our own code.

To create proxy classes create a new directory and within that directory issue the following command line commands (each should be entered on a single line):

```
WSDL /namespace:MyClientManagedServiceNS
 http://localhost/Ogsi.Container/services/transient/MyClientManagedService.asmx
```

```
WSDL /namespace:MyClientManagedServiceFactoryNS
 http://localhost/Ogsi.Container/services/persistent/MyClientManagedServiceFactory.asmx
```

This will create proxy classes called `MyClientManagedService.cs` and `MyClientManagedServiceFactory.cs` each within their own namespace to avoid name clashes with common types.

We can now write a client that uses the factory and the service instances it creates. The code for the client is (save this in a file called `MyTransClient.cs`):

```
using System;
using System.Xml;

using MyClientManagedServiceNS;
using MyClientManagedServiceFactoryNS;

class MyTransClient
{

   static void Main()
   {
      // Create a factory service object
      MyClientManagedServiceFactory factoryService =
                              new MyClientManagedServiceFactory();

      // Change the URL of the factory service to be the correct
      // container managed instance we deployed. Note the
      //correspondence between instanceId and the name attribute.

      factoryService.Url =
"http://localhost/ogsa/services/MyClientManagedFactoryServiceFactory.
asmx?instanceId=MyFirstFactory"

      // Construct XML creation element this is used to create
      // the service instance
      XmlDocument doc = new XmlDocument();
      XmlElement xmlParams = doc.CreateElement("initialValue");
      XmlNode    val       = doc.CreateTextNode("Ciao");
      xmlParams.AppendChild(val);

      // Construct a creation object
      MyClientManagedServiceFactoryNS.CreationType creation;
      creation = new CreationType();
      creation.terminationTimeSpecified = false;
      creation.serviceParameters = xmlParams;
```

```
      // Create a service using the factory
      MyClientManagedServiceFactoryNS.LocatorType locator;
      locator = factoryService.createService( creation );

      Console.WriteLine( "Have a service the locator is:\n"
        + locator.handle[0] );

     // Create service object
     MyClientManagedService service = new MyClientManagedService();

     // Tell this service object the location of the service
     // instance
     service.Url = locator.handle[0];

     // Use the service object
     Console.WriteLine( service.hello( "Mike"   ) );
     Console.WriteLine( service.hello( "Daragh" ) );
     Console.WriteLine( service.hello( "Ally"   ) );

     // Destroy the transient service when finished
     Service.destroy();
   }
}
```

There are several points to note about the code:

- We construct an XML element called `xmlParams` that contains the data required by the factory creator class. The schema of this element is specific to this particular factory, other factories can use different schema for their creation service parameters.
- The creation parameters XML element is bundled inside a `CreationType` object that is passed as input to the factory's `createService()` method.
- The `Url` property of all transient services proxies must be set before any methods of the proxy are called. The `Url` property must be set to the handle returned by the factory via the `LocatorType` object.
- The transient service can be destroyed when the client is finished with it by calling the `destroy()` method.

Compile this file along with `MyClientManagedService.cs` and `MyClientManagedServiceFactory.cs` as shown in the following command line:

```
csc /out:FactoryClient.exe MyTransClient.cs MyClientManagedService.cs
MyClientManagedServiceFactory.cs
```

This will produce an executable called `FactoryClient.exe` that will use the factory to create a new instance of the transient Grid Service and then call that service's `hello()` method several times to producing the output shown below:

```
Have a service the locator is:
http://localhost/Ogsi.Container/services/transient/MyClientManagedService.asmx?instanceId
=instance-141776660--214136689-1040913524821511526894621964229651231731 37
Ciao Mike 1
Ciao Daragh 2
Ciao Ally 3
```

9 Service data

The concept of service data is an important part of OGSI. The `GridServiceSkeleton` base class provides a collection for service data via the `InstanceServiceData` property. Through this collection the service developer can manipulate the service data in many ways. This section describes how the developer can manipulate the service data.

9.1 Default service data elements

All services will have several service data elements by default. These are defined by the OGSI specification and are implemented in the `GridServiceSkeleton` base class. The default service data elements are:

- interface
- factoryLocator
- gridServiceHandle
- gridServiceReference
- findServiceDataExtensibility
- setServiceDataExtensibility
- terminationTime
- serviceDataName

Other OGSI portTypes define a number of other default service data elements.

9.2 Service data set

The `GridServiceSkeleton` base class provides a protected member variable through which service implementations can access the service data collection. This member variable is called `serviceDataSet_` and is of type `ServiceDataSet`. There is also a property called `InstanceServiceDataSet` for use by clients of the `GridServiceSkeleton` object. The service data set is a container for service data elements. Each service data element is referenced by an XML qualified name (consisting of a local name and a namespace). The `ServiceDataSet` class provides methods to create new service data elements, add service data elements to the collection, access service data elements from the collection as well as several others. See the class documentation for full details of the class.

New service data elements can be added to the collection with code similar to the following:

```
// Construct a qualified name for the service data element
System.Xml.XmlQualifiedName sdeName =
  new System.Xml.XmlQualifiedName( "mySdeName",
                                   "MySdeNamespace" );

// Create a new service data element
ServiceData sde;
sde = serviceDataSet_.Create( sdeName );
```

```
// . . . code to populate the ServiceData object goes here

// Add service data element to the service data set
serviceDataSet_.Add( sde );
```

The indexer property of the `ServiceDataSet` can be used to access the `ServiceData` objects by the service data element's qualified name as show here:

```
// Construct the qualified name for the service data element
System.Xml.QualifiedName terminationTimeQName;
terminationTimeQName =
  new System.Xml.QualifiedName("terminationTime"
                               CoreConstants.OGSI_NAMESPACE_URI);

// Obtain the ServiceData object using the indexer
ServiceData sd = serviceDataSet_[ terminationTimeQName ];

// can now use sd to access and alter the termination time
// service data element
```

9.3 The `ServiceData` class

The `ServiceData` object is used to store the properties and values of the service data elements. For full details of the `ServiceData` class see the class documentation – this section provides an introduction to some of the mean features of the class.

9.3.1 Service data element properties

There are a number of standard OGSI properties that are associated with a service data element. These are listed below along with their default values and corresponding property of the `ServiceData` class.

OGSI property	Default value	`DataService` property
Minimum occurs	0	`minOccurs`
Maximum occurs	1	`maxOccurs`
Mutability	static	`mutability`
Modifiable	false	`modifiable`
Nillable	false	`nillable`

9.3.2 Service data values

Service data elements can have zero or more values. It is important to initialise a `ServiceData` object with details of the values that can be stored within it. For example, the following code configures a `ServiceData` object to store values of type `MyClass`:

```
// Assume there is a ServiceData object called sd

// Specify that the ServiceData objects will store values of
// type MyClass
```

```
sd.ValuesType = typeof(MyClass);
```

Values can be added to the `ServiceData` using the `AddValue()` method. For example:

```
MyClass myObject = new MyClass();
sd.AddValue( myObject );
```

There are several other methods and properties defined on the `ServiceData` class to manipulate the values of a service data element. See the class documentation for more details.

9.3.3 Using a callback to determine service data values dynamically

In some instances it is preferably to calculate the service data values dynamically as and when they are needed rather than storing a collection of values in the `ServiceData` object. This can be easily done by providing a callback class that the `ServiceData` class will use to determine the service data values whenever they are requested.

The callback class must implement the `IServiceDataValuesCallback` interface. This interface specifies a single read-only property called `ServiceDataValues` that returns an array of objects that are the service data values. For example:

```
class MyCallbackClass : IServiceDataValuesCallback
{
  public object[] ServiceDataValues
  {
    get
    {
       // code to create an object array of values goes here
       return result;
    }
  }
}
```

To specify that the callback mechanism should be used to determine the service data values the `Callback` property of the `ServiceData` class must be set to the instance of the callback class. For example:

```
MyCallbackClass callbackInstance = new MyCallbackClass();
sd.Callback =  callbackInstance;
```

It is important to note that this mechanism for dynamic service data value creation can only be used when the service data element is non-modifiable.

9.3.4 Lifetime attributes

Service data values can have lifetime attributes associated with them. To add a service data value with specific lifetime values to a `ServiceData` object the overloaded `AddValue()` method can be used. There is a version of the method that takes lifetime values as well as the value object itself. For example:

```
// Create value object
MyClass myObject = new MyClass();

// Create lifetime objects
System.DateTime goodFrom = System.DateTime.Now;
```

```
System.DateTime goodUntil = Ogsi.Core.Util.GridDateTime.INFINITY;
System.DateTime availableUntil =
                            Ogsi.Core.Util.GridDateTime.INFINITY;

sd.AddValue( myObject,
             goodFrom,
             goodUntil,
             availableUntil );
```

If no lifetime values are specified the `goodFrom` attribute defaults to the current time and the `goodUntil` and `availableUntil` attributes default to `INFINITY`.

It is possible to obtain the lifetime attributes associated with the values of a `ServiceData` object using the `GetLifetimes()` method. This method returns an array of `ServiceDataAttributes` objects each corresponding to a service data value. The `ServiceDataAttributes` class has properties to get and set the lifetime attributes and also get and set the value to which the lifetime attributes are associated.

See the class documentation for classes `ServiceData` and `ServiceDataAttributes` for more details.

9.3.5 Service data evaluators

The Grid Service findServiceData and setServiceData operations are implemented by the `GridServiceSkelton` base class. By default the findServiceData operation supports the queryByServiceDataNames query expression and the setServiceData operation supports the setByServiceDataNames and deleteByServiceDataNames query expressions.

It is possible to add new query expressions to these operations. To add support for a new query expression you must implement a new query evaluator to evaluate the query. Query evaluators must implement the `IExpressionEvaluator` interface. The evaluator must then be added to the query engine. The `GridServiceSkeleton` base class has a protected member variable `queryEngine_` of type `Ogsi.ServiceData.Query.QueryEngine`. This class has a method `RegisterEvaluator()` that can be used to add a query evaluator to the query engine. For examples of query evaluators see the implementations of `QueryByServiceDataNamesEvaluator` and `SetByServiceDataNamesEvaluator` both in the `Ogsi.ServiceData.Query` namespace.

10 PortTypes

A large portion of [OGSI-Spec] addresses the issue of services providing multiple portTypes, and allowing services to inherit from multiple portTypes. In our approach, the `WebMethodAttribiute`-marked methods in the proxy classes essentially represent the most-derived portType for a service, and it is with this that the clients communicate. At present, it is only possible to present this most-derived portType to the client. It is possible to alter the properties and namespaces of the exposed operations by using the `WebServiceAttribute`, `SoapDocumentMethodAttribute` and `SoapRpcMethodAttribute` classes, and others, provided by the .NET framework. See the .NET user documentation for details.

We address the issue of portType re-use in the following manner. When developing a service implementation, service methods may be placed on the class you derive from `GridServiceSkeleton`, like so:

```
public class MyService : GridServiceSkeleton
{
      public string myMethod()
      {
            this.InstanceServiceDataSet[someQName] = 10;
            return "hello";
      }
}
```

With this method, access to the service's `ServiceDataSet` is via the `InstanceServiceData` property inherited from `GridServiceSkeleton`. Adding methods in this manner essentially means adding methods to a base service portType.

In other circumstances, it may be desirable to use functionality developed elsewhere to provide operations, or to logically group related functionality into single portTypes, which you then wish to aggregate on your service. We provide means of doing this using the `Ogsi.Core.OgsiPortTypeAttribute` and `Ogsi.Core.OperationProviderBase` classes and the `Ogsi.Core.IOperationProvider` interface.

10.1 OgsiPortTypeAttribute

We use as an example a class `SomeUsefulClass`, with a method `SomeUsefulMethod`. It is desired to expose this method as an operation on your service. The simplest way to do this is as follows:

```
[OgsiPortType( typeof( MyAssembly.SomeUsefulClass ),
               "http://mydomain.com/myNameSpace",
               "UsefulPortType" );
public class MyService : GridServiceSkeleton
{
}
```

The first parameter in the attribute is the type of your provider class. The second parameter is the URI of the namespace for this new portType. The third parameter is the name of the portType. The latter two parameters do not have an effect at the moment but are included for possible future work.

When the `MyService` class is instantiated by the container, instances of every class defined

by its `OgsiPortTypeAttributes` are created and associated with the service instance. When developing proxy methods for provider-based methods, the `CallMethodOnPortType` method of `GridServiceInstanceAspProxy` is used:

```
[WebMethod]
public int SomeUsefulMethod()
{
  return CallMethodOnPortType(
    "MyAssembly.SomeUsefulClass",
    "SomeUsefulMethod",
    (object[]) null
  );
}
```

The provider classes can be given access to the `ServiceDataSet` of the service in the following manner. If the provider classes are found to implement `IOperationProvider`, the `ServiceInstance` property defined by this interface is initialised with a reference to the instance of `MyService` that it is associated during initialisation.

We provide a basic implementation of `IOperationProvider` called `Ogsi.Core.OperationProviderBase` for convenience. It is recommended that you always base your portType implementations on this class.

The provider class may now access the service's `ServiceDataSet` by obtaining a reference from the reference it maintains to the service class:

```
public class SomeOtherUsefulClass : OperationProviderBase
{
  public int SomeOtherUsefulMethod()
  {
    int i;
    // process
    ...
    this.ServiceInstance.InstanceServiceData[someQName] = i;
    return i;
  }
}
```

11 Notification

The OGSI specification defines behaviours that allow clients to subscribe to changes in the values of certain service data elements. MS.NETGrid-OGSI implements this portion of the specification as of release 1.2. A demonstration project is contained with this distribution that shows how the notification functionality can be used. See the `StockValueService` demonstrator for details.

11.1 Making a service data element notifiable

You may wish only to expose certain service data elements in your service available for notification subscriptions. The `Ogsi.ServiceData.ServiceData` class has a property called `IsNotifiable` for this purpose. The default value of this property is `false`.

An example of use:

```
// Construct a qualified name for the service data element
System.Xml.XmlQualifiedName sdeName =
  new System.Xml.XmlQualifiedName( "mySdeName",
                                   "MySdeNamespace" );

// Create a new service data element
ServiceData sde;
sde = serviceDataSet_.Create( sdeName );
sde.IsNotifiable = true;
```

When the `IsNotifiable` property is set to true:

- Clients may subscribe for notifications to changes in the value(s) of this service data element
- The service data elements will appear in the values of the `ogsi:notifiableServiceDataName` service data element on the NotificationSourcePortType of the service.

11.2 Creating a Source Service

A Grid Service can act as a source of notification messages by using the `Ogsi.Core.Notification.NotificationSourcePortType` class via the `OgsiPortTypeAttribute` as described above. This means that any notifiable service data (in the sense explained in Section 10.1 above) in the service data set of this service may be subscribed to for changes in their values.

An example of the use of `NotificationSourcePortType` is:

```
[OgsiPortType( typeof( NotificationSourcePortType ),
  OgsiConstants.OGSI_NAMESPACE_URI, "NotificationSource" )]
public class MyNotificationSourceServiceImpl :
    GridServiceSkeleton
{
  // Implementation code here.
}
```

You will need to add code to the proxy for you service that exposes the `Subscribe` method

of `NotificationSourcePortType` in the following manner:

```
public class MyNotificationSourceService :
    GridServiceSkeletonInstanceAspProxy
{
    [WebMethod]
    public subscribeResponse subscribe(
        XmlElement subscriptionExpression,
        LocatorType sink,
        TerminationTimeType expirationTime
        )
    {
        FaultType fault = null;
        subscribeResponse response;
        object [] args = new object [] {
                    subscriptionExpression,
                    sink,
                    expirationTime,
                    fault };

        response =
          ( subscribeResponse ) CallMethodOnPortType(
          "Ogsi.Core.Notification.NotificationSourcePortType",
          "Subscribe",
          args );
          CheckFault( fault );
          return response;
    }
}
```

You must alert the MS.NETGrid-OGSI container when you have changed service data values. This can be performed via the `NotifyChanged()` method on the `Ogsi.ServiceData.ServiceData` class. An example of this is:

```
ServiceData someServiceDataElement = InstanceServiceDataSet[
      new XmlQualifiedName( "myServiceDataElement",
      "http://someNameSpace.com") ]
SomeServiceDataElement.Value = newValue;
SomeServiceDataElement.NotifyChanged();
```

Once `NotifyChanged()` has been called, the container automatically alerts all subscribed parties to the change in values.

11.3 Creating a sink service

We provide the abstract `NotificationSinkPortType` class with the MS.NETGrid software. Since the behaviour of notification sinks upon receiving notification messages is not defined by [OGSI-Spec], you must inherit from this class and override the `deliverNotification()` method to carry out application specific processing.

The intended recipient of a notification message must be able to listen for and process HTTP requests and the SOAP messages contained within them. The .NET framework generally relies on IIS for the processing of Web requests, which means that, should you wish an application other than a Web service to receive notification messages, you would have to implement your own in-process HTTP listener. Indeed, this is the approach taken by

[Virginia-Impl]. We have not provided such a capability in our release.

11.3.1 The debug notification service

We provide as part of the distribution a Web service that will take notification messages and write them to a file. This service is to be found under the `Ogsi.Container/services/debug/` directory. The service also allows you to obtain the last message that it received via the `GetLastMessage()` operation. This service is used by the Notification monitor client application that is included as part of the distribution. The service is located in the `<Install_dir>/Ogsi.Container/debug` directory.

11.4 Writing clients of notification sources

A list of notifiable service data names for a service that implements `NotificationSourcePortType` is obtained by getting the value of the `ogsi:notifiableServiceDataName` service data element in the usual manner i.e. using the QueryHelper class to generate a find query and then using the `findServiceData` method of the client side proxies to obtain the values.

Once this list has been obtained, a client can subscribe to notifications to changes to one of these service data elements via the `subscribe` method of NotificationSubscriptionPortType. In the following example, the `notifiableNames` collection contains the names of the service data elements that are to be subscribed to. An ArrayList containing these names is

```
ArrayList notifiableNames;
// The service client side proxy, as generated by WSDL.exe
NotificationSourceServcice notificationSourceService = new
   NotificationSourceService();
.. ..
XmlElement subscriptionExpression =
QueryHelper.GenerateNotificationSubscriptionByNamesExpression(
                 notifiableNames );

// Set up the locator for the service
// SetUpSinkLocator is some code somewhere that sets up
// a new Locator with info about the sink.
LocatorType sinkLocator = SetUpSinkLocator();

// Termination time for subscription instance
TerminationTimeType termTime = new TerminationTimeType();
// Set the term time to contain appropriate value
.. ..

// Set carry out the subscription
subscribeResponse response
    = notificationSourceService.subscribe(
          subscriptionExpression,
          sinkLocator,
          termTime
      );
// response gives you access to the subscription instance
// created by the request.
```

The `subscribeResponse` object that is returned by the `subscribe` operation may be used to manage the subscription. To de-register interest in changes to the values of the subscribed service data elements, call the `destroy` method of the subscription service instance.

The Notification source monitor client included as part of the distribution (in `<Install_dir>/NotificationSourceMonitor`) demonstrates more fully how to write client applications that utilise notification capabilities.

12 Persisting services

Release 2.0 of the MS.NETGrid-OGSI container allows you to persist stateful services to permanent storage on container shutdown. This means that all current service instances running in the container will be available with their state intact when the container is restarted. In order to do this, there are a number of restrictions on the way you write your services:

- You must make sure all classes that are associated with your service (skeleton, portType implementations, helper classes) are serializable. This means that all relevant classes should have the [`Serializable`] attribute, and optionally should implement the `ISerializable` interface. Further details of the .NET serialization APIs are available in the .NET framework documentation.

- You must configure the container to use either the filesystem or an SQL Server database to persist service instances.

12.1 Configuring the container

Configuring the container for persistence is quite easy. In the `Web.config` file, there is a section that looks like the following in the `containerProperties` section.

```
<!-- What class is used to store the services on container shutdown
        <add key="permanentStorageType"
value="Ogsi.Container.Persistence.FileSystemServicePermanentStorage,
Ogsi.Container"/>
        <add key="permanentStorageDirectory"
value="D:\MS.NETGrid\Ogsi.Container\bin\servicePermanentStorage" />

        OR


        <add key="permanentStorageType"

value="Ogsi.Container.Persistence.SqlServerServicePermanentStorage,
Ogsi.Container"/>
        <add key="connectionStringForServiceStorage"
                value="workstation id=MY_WS_ID;packet
size=4096;integrated security=SSPI;data source=MY_WS_ID;persist
security info=False" />
          -->
```

If you wish to persist services to disk, uncomment the first set of properties. The `permanentStorageType` property should be left unaltered, but you can change the `permanentStorageDirectory` value to the appropriate value, depending on your installation.

If you wish to persist services to an SQL server database, first you must create the appropriate database on your SQL server. We provide a script called `createPersistenceDB.sql`, in `<install_dir>/Ogsi.Container/`. Next you need to change the value of the connection string property in the configuration file. See your SQL Server user documentation for details.

13 Example services

As an example of using portTypes, serviceData and persistent and transient services, we provide a pair of demonstration services. These are located in the <*install_dir*>/MS.NETGrid/Ogsi.Container/demoservices container.

The basicService subdirectory contains a persistent Grid Service that implements the GridService portType, a transient version of this service and a factory for creating instances of the transient service. The proxy code for these services is located in Ogsi.Container/services/persistent and Ogsi.Container/services/transient. These services do not expose any additional functionality, but do demonstrate the programming model that this document describes.

The counter subdirectory contains a more interesting example. The transient service, located in CounterServiceSimple.cs, exposes all the functionality of the GridService portType, inherited from GridServiceSkeleton. In addition, Increment and Decrement methods are provided. A serviceData element called count is defined with its own namespace. The Increment and Decrement methods are used to alter the value of this serviceData element. In addition, the value of count may be set using the setServiceData operation provided by the GridService portType.

The deployment for the transient counter service and the persistent counter factory service may be examined in <*install_dir*>/MS.NETGrid/Ogsi.Container/Web.config.

The counter factory demonstrates the use of previously existing portTypes via the OgsiPortTypeAttribute attribute. The CounterFactoryCreator class demonstrates the use of the IFactoryProvider interface. The CreateServiceObject() method takes an XML element with that specifies the initial value of the counter.

The notification demonstrator service has already been mentioned in Section 11 above.

14 Glossary

For convenience of reference we include the following glossary.

.NET – The Microsoft .NET platform, including the .NET SDK (Software Development Kit), associated development tools such as Visual Studio .NET [http://www.microsoft.com/net] and the Common Language Runtime.

Common Language Runtime – a runtime environment that executes Microsoft Intermediate Language (MSIL).

MSIL – The .NET bytecode, a low-level assembler-like language executed by the Common Language Runtime.

Managed code – Code that runs on the .NET CLR.

IIS – Microsoft Internet Information Server, a web server offering communication over a variety of internet protocols.

ISAPI – An Application Programming Interface (API) that allows applications to use the network services provided by IIS. Commonly used to provide HTTP filters that respond to all HTTP requests on the server.

AppDomain – .NET allows the partitioning of a single Operating System process into a number of AppDomains, which are essentially memory safe areas within the process. If the code in an AppDomain crashes, other AppDomains within the process are unaffected. This concept has some performance advantages as using AppDomains avoids the overheads associated with starting a new process.

.NET Remoting – The mechanism by which .NET allows Remote Procedure Call (RPC) between AppDomains and between remote machines.

ASP – Active Server Pages, a Microsoft technology that works in conjunction with IIS to host Web Applications.

ASP.NET – The .NET version of ASP, with extensions for Web Services and Enterprise Applications.

C# - Pronounced C-Sharp, a new Microsoft programming language that takes advantage of .NET platform features.

Assembly – A collection of managed code, which may form an executable or a library, and may be distributed across a number of physical files. Assemblies facilitate the separation of logical and physical resources.

OGSA – The Open Grid Services Architecture, which is a set of proposed architectures and protocols for Grid computing [Anatomy].

OGSI – The Open Grid Services Infrastructure, a specification of behaviours and properties of Grid Services [Physiology, OGSI-Spec].

OGSA-DAI – OGSA Data Access and Integration, Grid Services framework for seamless access and integration of data using the OGSA paradigm [OGSA-DAI].

Globus – The Globus toolkit. A standard toolkit of Grid software. Version 3 will contain an implementation of OGSI in Java [http://www.globus.org].

WSDL – The Web Services Description Language [http://www.w3.org/TR/wsdl].

Web Services – Refers to network-enabled software capabilities accessed using common Internet protocols such as HTTP and SOAP and described using WSDL.

Grid Services – Web Services conforming to the OGSI specification.

Grid Service Instance – A network accessible instance of an OGSI-compliant service.

PortType – A set of operations supported by a Web or Grid Service.

ServiceData – An XML-based mechanism that allows a client to query the state of a service instance in a flexible and extensible manner.

ServiceDataElement – A particular element of serviceData, identified by a name and a value

Tomcat – A container for Java-based Web Applications [http://jakarta.apache.org].

AXIS – A Web Application running under Tomcat, which provides Web Services functionality. [http://xml.apache.org].

Programming Model – A set of procedures and APIs used to develop an application in a given domain

References

Documents referenced in the text, and other related documents, include the following.

[MS.NETGridOgsiDesignOverview] MS.NETGrid OGSI Implementation Design Overview (MS.NETGridOgsiDesignOverview-v1.0), *D.Byrne*, EPCC, June 16th 2003.

[MS.NETGrid-Proj-Def] Project Definition for a Collaboration Between Microsoft and EPCC (MS.NetGrid-ProjDef-V2.0), *M. Jackson*, EPCC, April 25th 2003.

[OGSI-Spec] S. Tuecke, K. Czajkowski, I. Foster, J. Frey, S. Graham, C. Kesselman, T. Maquire, T. Sandholm, D. Snelling, P. Vanderbilt. Open Grid Services Infrastructure (OGSI) Version 1.0, June 27th 2003. http://forge.gridforum.org/projects/ogsi-wg

[Virginia-Impl] – OGSI.NET: An OGSI-compliant Hosting Container for the .NET Framework, Grid Computing Group, University of Virginia. WWW site: http://www.cs.virginia.edu/~humphrey/GCG/ogsi.net.html.

[GT3] – Globus Toolkit version 3 and OGSI, Available at http://www.globus.org/ogsa

[Physiology] – The Physiology of the Grid (Draft 2.9), *I. Foster, C. Kesselman, J.Nick, S. Tuecke,* Available at http://www.gridforum.org.org/ogsa-wg

[Anatomy] – The Anatomy of the Grid, *I. Foster, C. Kesselman, S. Tuecke,* Available at http://www.gridforum.org/ogsa-wg

[OGSA-DAI] – Open Grid Service Architecture, Database Access and Integration http://www.ogsadai.org.uk

[WSDL-Spec] Web Services Description Language 1.1, http://www.w3.org/TR/2001/NOTE-wsdl-20010315

[AXIS] – The Apache Axis SOAP Engine, http://ws.apache.org/axis/